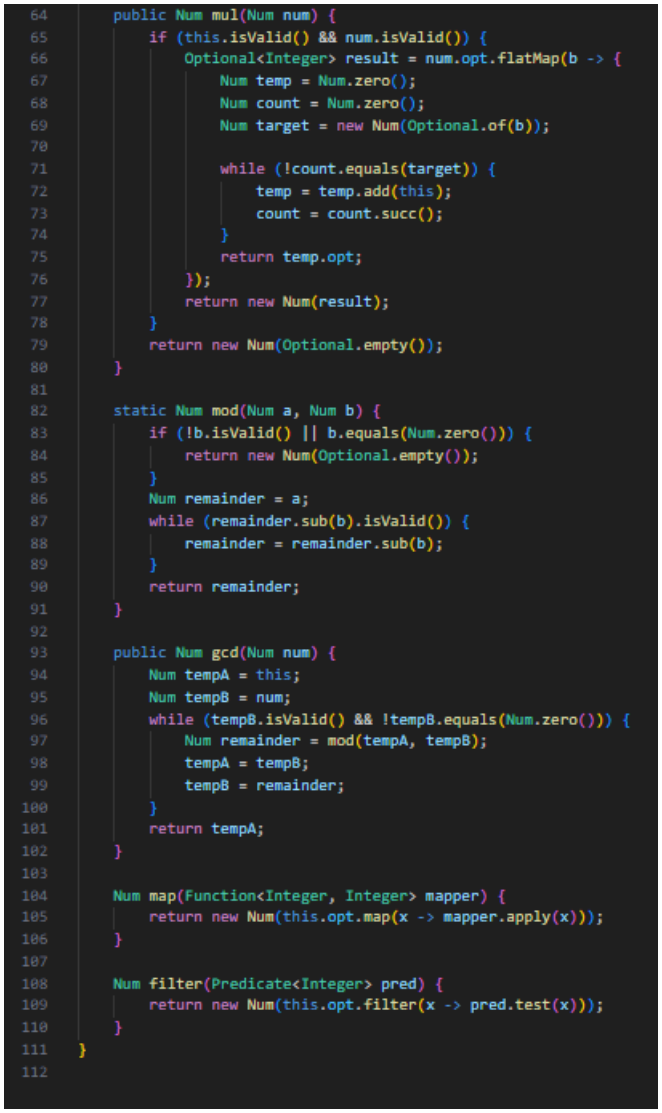
```java
import java.util.List;

class Main {
    static double simulate(int seed, int n) {
        Circle circle = new Circle(new Point(0, 0), 1.0);

        List<Double> random =
            Rand.randRange(seed, x -> (2.0) * x / (Integer.MAX_VALUE - 1) - 1.0)
                .limit(n * 2)
                .toList();

        long count = 0;
        for (int i = 0; i < random.size(); i += 2) {
            double x = random.get(i);
            double y = random.get(i + 1);
            Point p = new Point(x, y);

            if (circle.contains(p)) {
                count++;
            }
        }

        return 4.0 * count / n;
    }
}
```

```java
import java.util.Random;
import java.util.function.Function;
import java.util.stream.Stream;

class Rand<T> {
    private final int seed;
    private final Function<Integer, T> func;

    private Rand(int seed, Function<Integer, T> func) {
        this.seed = seed;
        this.func = func;
    }

    static Rand<Integer> of(int seed) {
        return new Rand<Integer>(seed, x -> x);
    }

    static <R> Rand<R> of(int seed, Function<Integer, R> func) {
        return new Rand<>(seed, func);
    }

    public T get() {
        return this.func.apply(this.seed);
    }

    public Rand<T> next() {
        return new Rand<T>(new Random(this.seed).nextInt(Integer.MAX_VALUE),
                           this.func);
    }

    public Stream<T> stream() {
        return Stream.iterate(this, rand -> rand.next())
            .map(random -> random.get());
    }

    static <R> Stream<R> randRange(int seed, Function<Integer, R> pred) {
        return Rand.of(seed).stream().map(x -> pred.apply(x));
    }

    public <R> Rand<R> map(Function<? super T, ? extends R> mapper) {
        Function<Integer, R> newFunc = this.func.andThen(mapper);
        return Rand.of(this.seed, newFunc);
    }

    public <R> Rand<R> flatMap(Function<? super T, Rand<R>> mapper) {
        Function<Integer, R> newFunc = seed -> {
            T t = this.func.apply(seed);
            Rand<R> randR = mapper.apply(t);
            return randR.get();
        };
        return Rand.of(this.seed, newFunc);
    }

    @Override
    public String toString() {
        return "Rand";
    }
}
```

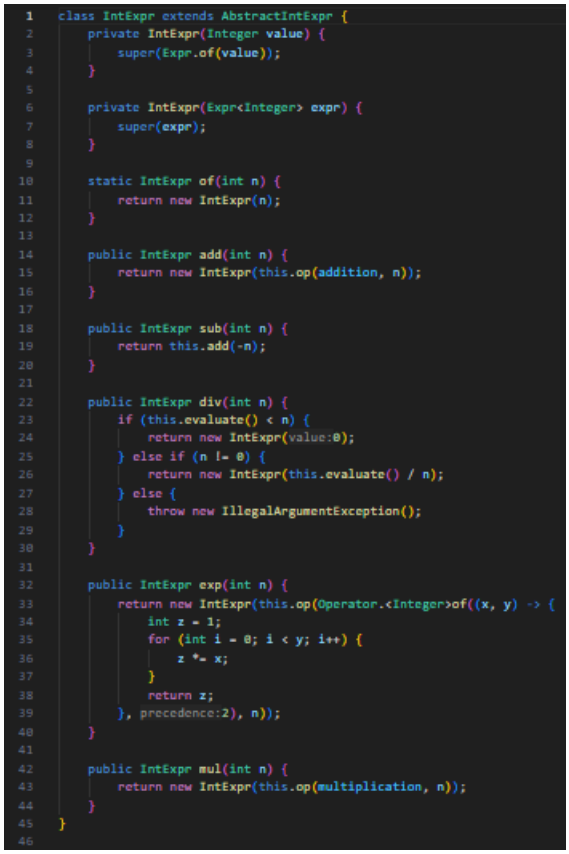
```java
import java.util.Optional;
import java.util.function.Function;
import java.util.function.Predicate;

class Num extends AbstractNum<Integer> {
    private Num(Integer val) {
        super(val);
    }

    public Num(Optional<Integer> opt) {
        super(opt);
    }

    private Num(AbstractNum<Integer> num) {
        super(num.opt);
    }

    static Num zero() {
        return new Num(AbstractNum.zero());
    }

    static Num one() {
        return zero().succ();
    }

    static Num of(int i) {
        if (AbstractNum.valid.test(i)) {
            return new Num(i);
        }
        return new Num(Optional.empty());
    }

    public Num succ() {
        if (this.isValid()) {
            return new Num(this.map(AbstractNum.s));
        } else {
            return this;
        }
    }

    Num add(Num n) {
        if (n.isValid() && this.isValid()) {
            Num count = zero();
            Num sum = this;
            while (!count.equals(n)) {
                sum = sum.succ();
                count = count.succ();
            }
            return sum;
        } else {
            return new Num(Optional.empty());
        }
    }

    Num sub(Num n) {
        if (n.isValid() && this.isValid()) {
            Num negatedN = new Num(n.opt.map(AbstractNum.n));
            Num result = negatedN.add(this);
            return new Num(result.filter(AbstractNum.valid));
        }
        return new Num(Optional.empty());
    }
```



```
    public Num mul(Num num) {
        if (this.isValid() && num.isValid()) {
            Optional<Integer> result = num.opt.flatMap(b -> {
                Num temp = Num.zero();
                Num count = Num.zero();
                Num target = new Num(Optional.of(b));

                while (!count.equals(target)) {
                    temp = temp.add(this);
                    count = count.succ();
                }
                return temp.opt;
            });
            return new Num(result);
        }
        return new Num(Optional.empty());
    }

    static Num mod(Num a, Num b) {
        if (!b.isValid() || b.equals(Num.zero())) {
            return new Num(Optional.empty());
        }
        Num remainder = a;
        while (remainder.sub(b).isValid()) {
            remainder = remainder.sub(b);
        }
        return remainder;
    }

    public Num gcd(Num num) {
        Num tempA = this;
        Num tempB = num;
        while (tempB.isValid() && !tempB.equals(Num.zero())) {
            Num remainder = mod(tempA, tempB);
            tempA = tempB;
            tempB = remainder;
        }
        return tempA;
    }

    Num map(Function<Integer, Integer> mapper) {
        return new Num(this.opt.map(x -> mapper.apply(x)));
    }

    Num filter(Predicate<Integer> pred) {
        return new Num(this.opt.filter(x -> pred.test(x)));
    }
}
```

```java
import java.util.Optional;

class Fraction extends AbstractNum<Frac> {
    private Fraction(Optional<Frac> frac) {
        super(frac);
    }

    static Fraction of(Integer num, Integer denom) {
        if (valid.test(num) && valid.test(denom) &&
            !Num.of(denom).equals(Num.zero())) {
            return new Fraction(Optional.<Frac>of(Frac.of(Num.of(num), Num.of(denom))));
        }
        return new Fraction(Optional.empty());
    }

    Fraction add(Fraction frac) {
        if (this.isValid() && frac.isValid()) {
            Optional<Num> a = this.opt.map(x -> x.t());
            Optional<Num> b = this.opt.map(x -> x.u());
            Optional<Num> c = frac.opt.map(x -> x.t());
            Optional<Num> d = frac.opt.map(x -> x.u());

            Optional<Num> ad = a.flatMap(x -> d.map(y -> y.mul(x)));
            Optional<Num> bc = b.flatMap(x -> c.map(y -> y.mul(x)));
            Optional<Num> bd = b.flatMap(x -> d.map(y -> y.mul(x)));
            Optional<Num> adbc = ad.flatMap(x -> bc.map(y -> y.add(x)));

            Optional<Frac> value =
                adbc.flatMap(x -> bd.map(y -> Frac.of(x, y)));
            return new Fraction(value);
        }
        return new Fraction(Optional.empty());
    }

    Fraction sub(Fraction frac) {
        if (this.isValid() && frac.isValid()) {
            Optional<Num> a = this.opt.map(x -> x.t());
            Optional<Num> b = this.opt.map(x -> x.u());
            Optional<Num> c = frac.opt.map(x -> x.t());
            Optional<Num> d = frac.opt.map(x -> x.u());

            Optional<Num> ad = a.flatMap(x -> d.map(y -> y.mul(x)));
            Optional<Num> bc = b.flatMap(x -> c.map(y -> y.mul(x)));
            Optional<Num> bd = b.flatMap(x -> d.map(y -> y.mul(x)));
            Optional<Num> adbc = ad.flatMap(x -> bc.map(y -> x.sub(y)));

            if (!adbc.equals(Optional.of(new Fraction(Optional.empty())))) {
                Optional<Frac> value =
                    adbc.flatMap(x -> bd.map(y -> Frac.of(x, y)));
                return new Fraction(value);
            }
        }
        return new Fraction(Optional.empty());
    }

    Fraction mul(Fraction frac) {
        if (this.isValid() && frac.isValid()) {
            Optional<Num> a = this.opt.map(x -> x.t());
            Optional<Num> b = this.opt.map(x -> x.u());
            Optional<Num> c = frac.opt.map(x -> x.t());
            Optional<Num> d = frac.opt.map(x -> x.u());

            Optional<Num> ac = a.flatMap(x -> c.map(y -> y.mul(x)));
            Optional<Num> bd = b.flatMap(x -> d.map(y -> y.mul(x)));

            Optional<Frac> value = ac.flatMap(x -> bd.map(y -> Frac.of(x, y)));
            return new Fraction(value);
        }
        return new Fraction(Optional.empty());
    }
}
```

```java
import java.util.function.BinaryOperator;

class Operator<T> implements Comparable<Operator> {
    private final BinaryOperator<T> binOp;
    private final int precedence;

    private Operator(BinaryOperator<T> binOp, int precedence) {
        this.binOp = binOp;
        this.precedence = precedence;
    }

    static <T> Operator<T> of(BinaryOperator<T> binOp, int precedence) {
        return new Operator<>(binOp, precedence);
    }

    public T apply(T a, T b) {
        return this.binOp.apply(a, b);
    }

    @Override
    public int compareTo(Operator other) {
        return Integer.compare(this.precedence, other.precedence);
    }

    @Override
    public String toString() {
        return "Operator of precedence " + this.precedence;
    }
}

```



```
class IntExpr extends AbstractIntExpr {
    private IntExpr(Integer value) {
        super(Expr.of(value));
    }

    private IntExpr(Expr<Integer> expr) {
        super(expr);
    }

    static IntExpr of(int n) {
        return new IntExpr(n);
    }

    public IntExpr add(int n) {
        return new IntExpr(this.op(addition, n));
    }

    public IntExpr sub(int n) {
        return this.add(-n);
    }

    public IntExpr div(int n) {
        if (this.evaluate() < n) {
            return new IntExpr(value:0);
        } else if (n != 0) {
            return new IntExpr(this.evaluate() / n);
        } else {
            throw new IllegalArgumentException();
        }
    }

    public IntExpr exp(int n) {
        return new IntExpr(this.op(Operator.<Integer>of((x, y) -> {
            int z = 1;
            for (int i = 0; i < y; i++) {
                z *= x;
            }
            return z;
        }, precedence:2), n));
    }

    public IntExpr mul(int n) {
        return new IntExpr(this.op(multiplication, n));
    }
}

```

```java
import java.util.Optional;

class Expr<T> {
    private final T value;
    private final Optional<Operator<T>> operator;
    private final Optional<Expr<T>> left;
    private final Optional<Expr<T>> right;

    private Expr(T value) {
        this.value = value;
        this.operator = Optional.empty();
        this.left = Optional.empty();
        this.right = Optional.empty();
    }

    private Expr(T value, Optional<Operator<T>> operator,
                 Optional<Expr<T>> left, Optional<Expr<T>> right) {
        this.value = value;
        this.operator = operator;
        this.left = left;
        this.right = right;
    }

    public Expr(Expr<T> expr) {
        this.value = expr.value;
        this.operator = expr.operator;
        this.left = expr.left;
        this.right = expr.right;
    }

    static <T> Expr<T> of(T value) {
        return new Expr<>(value);
    }

    public Expr<T> op(Operator<T> otherOperator, T otherValue) {
        Expr<T> otherExpr = Expr.of(otherValue);

        return this.operator
            .map(current -> {
                if (otherOperator.compareTo(current) > 0) {
                    return new Expr<>(this.value, Optional.of(otherOperator),
                                      Optional.of(this),
                                      Optional.of(otherExpr));
                } else {
                    return new Expr<>(
                        this.value, Optional.of(current), this.left,
                        Optional.of(new Expr<>(
                            otherExpr.value, Optional.of(otherOperator),
                            this.right, Optional.of(otherExpr))));
                }
            })
            .orElse(new Expr<>(this.value, Optional.of(otherOperator),
                               Optional.of(this), Optional.of(otherExpr)));
    }

    public Expr<T> op(Operator<T> operator, Expr<T> expr) {
        return new Expr<>(this.value, Optional.of(operator), Optional.of(this),
                          Optional.of(expr));
    }

    public T evaluate() {
        return this.operator
            .map(op
                 -> op.apply(left.orElseThrow().evaluate(),
                             right.orElseThrow().evaluate()))
            .orElse(value);
    }

    @Override
    public String toString() {
        return this.operator
            .map(op
                 -> "(" +
                    op.apply(this.left.orElseThrow().evaluate(),
                             right.orElseThrow().evaluate()) +
                    ")")
            .orElse("(" + this.value + ")");
    }
}
```

```
import java.util.function.Predicate;
import java.util.function.BinaryOperator;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.Optional;

class DnC <T , R> {
    private final Supplier<T> problem;
    private final Predicate<T> checkAtomic;
    private final Function<T, R> mapper;
    private final Optional<Function<T,Pair<Supplier<T>,Supplier<T>>>> transform;

    private DnC(Supplier<T> problem, Predicate<T> checkAtomic, Function<T, R> mapper) {

        this.problem = problem;
        this.checkAtomic = checkAtomic;
        this.mapper = mapper;
        this.transform = Optional.empty();

    }

    public DnC(Supplier<T> problem, Predicate<T> checkAtomic, Function<T, R> mapper,
       Optional<Function<T,Pair<Supplier<T>,Supplier<T>>>> transform) {

        this.problem = problem;
        this.checkAtomic = checkAtomic;
        this.mapper = mapper;
        this.transform = transform;
    }

    public static  <T , R> DnC<T, R> of(T prob, Predicate<T> checkAtomic, Function<T, R> mapper) {

        return new DnC<T, R>(() -> prob, checkAtomic, mapper);
    }

    public static  <T , R> DnC<T,R> of(T prob,  Predicate<T> checkAtomic, Function<T, R> mapper,
        Function<T,Pair<T,T>> transform) {

            Function<Pair<T,T>, Pair<Supplier<T>, Supplier<T>>> f  =
            pair -> Pair.of(() -> pair.first(), () -> pair.second());

            return new DnC<T, R>(() -> prob, checkAtomic, mapper, Optional.of(transform.andThen(f)));
    }

    public static <T , R> DnC<T, R> of(Supplier<T> problem, Predicate<T> checkAtomic,
        Function<T, R> mapper, Function<T, Pair<Supplier<T>, Supplier<T>>> transform) {

        return new DnC<T, R>(problem, checkAtomic, mapper, Optional.ofNullable(transform));
    }


    public void peek(Consumer<T> action) {

        action.accept(this.problem.get());
    }

    public DnC<T, R> left() {

        return this.left(this.problem.get());
    }
```

```
public DnC<T, R> left(T problemGet) {

    return Optional.of(problemGet)
        .filter(x -> !this.checkAtomic.test(x))
        .flatMap(prob -> this.transform.map(div -> {
            Pair<Supplier<T>, Supplier<T>> pair = div.apply(prob);
            return new DnC<T, R>(pair.first(), this.checkAtomic,
            this.mapper, this.transform);
        }))
        .orElse(this);
}

public DnC<T, R> right() {

    return this.right(this.problem.get());
}


public DnC<T, R> right(T problemGet) {

    return Optional.of(problemGet)
        .filter(x -> !this.checkAtomic.test(x))
        .flatMap(prob -> this.transform.map(div -> {
            Pair<Supplier<T>, Supplier<T>> pair = div.apply(prob);
            return new DnC<T, R>(pair.second(), this.checkAtomic,
            this.mapper, this.transform);

        }))
        .orElse(this);
}

public Optional<R> solve() {

    return this.solve(this.problem.get());
}

public Optional<R> solve(T problemGet) {

    boolean checking = checkAtomic.test(problemGet);
    return Optional.of(problemGet)
    .filter(x -> checking)
    .map(x -> mapper.apply(x));
}

public Optional<R> solve(BinaryOperator<R> combiner) {

    return this.solve(this.problem.get(), combiner);
}

public Optional<R> solve(T problemGet, BinaryOperator<R> combiner) {

    return this.solve(problemGet).or(
    () -> left(problemGet).solve(combiner)
    .flatMap(leftS -> right(problemGet).solve(combiner)
    .map(rightS -> combiner.apply(leftS, rightS)
    )));
}
```

```java
import java.util.List;
import java.util.Optional;

class SumList extends DnC<List<Integer>,Integer> {

    public SumList(List<Integer> i) {
        super(

        () -> i,
        x -> x.size() == 1,

        x -> x.get(0), Optional.of(list -> {
                int mid = (list.size() + 1) / 2;

                return Pair.of(
                () -> list.subList(0,mid),
                () -> list.subList(mid,list.size())
                );
        }));
    }
}
```

```java
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

public class Str {
    private final Function<Consumer<String>, String> str;

    private Str(Function<Consumer<String>, String> str) {
        this.str = str;
    }

    public static Str of(String value) {
        return new Str(x -> {
            x.accept("traced Str: " + value);
            return value;
        });
    }

    public static Str of(Supplier<String> supplier) {
        return new Str(x -> {
            String s = supplier.get();
            x.accept("traced Str: " + s);
            return s;
        });
    }

    public Str map(Function<String, String> mapper) {
        return new Str(x -> {
            String s = mapper.apply(this.str.apply(x));
            x.accept("traced map: " + s);
            return s;
        });
    }

    public Str flatMap(Function<String, Str> mapper) {
        return new Str(x -> {
            String s = mapper.apply(this.str.apply(x)).str.apply(x);
            x.accept("traced flatMap: " + s);
            return s;
        });
    }

    public Str join(String other) {
        return this.map(x -> x + other);
    }

    public Str join(Str other) {
        return this.flatMap(x -> other.map(y -> x + y));
    }

    public void run(Consumer<String> action) {
        action.accept(this.str.apply(x -> {}));
    }

    public void print() {
        this.run(x -> System.out.println(x));
    }

    public void trace() {
        this.trace(x -> System.out.println(x));
    }

    public void trace(Consumer<String> tracer) {
        String s = this.str.apply(tracer);
        tracer.accept(s);;
    }
}
```